Reinforcement Learning

Machine Learning 2018
Peter Bloem

the adaptive intelligent agent

The first thing we did, in the first lecture, when we first discussed the idea of machine learning, was **to take it offline**. We simplify the problem by assuming that we have a training set from which we learn a model in a single pass. We reduced the problem of adaptive intelligence to a single pass over a dataset by removing the idea of interacting with an outside world, and by removing the idea of continually learning and actin at the same time.

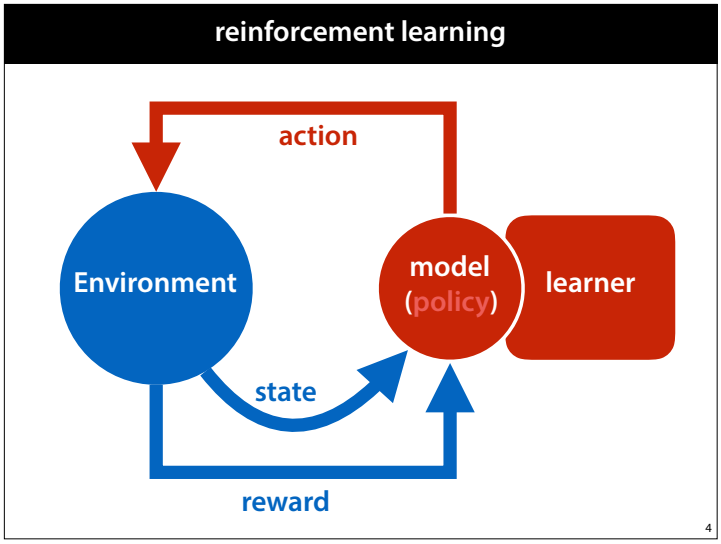
Sometimes those aspects can not be reduced away. In such cases we can use the framework of **Reinforcement Learning**. Reinforcement learning is the practice of training agents (e.g. robots) that interact with a dynamic world, and to train them to learn while they're interacting.

the plan

part 1:
Reinforcement Learning

Approaches:
- Random search,
- Policy Gradients,
- Q-Learning

part 2:
AlphaGo
AlphaZero
AlphaStar

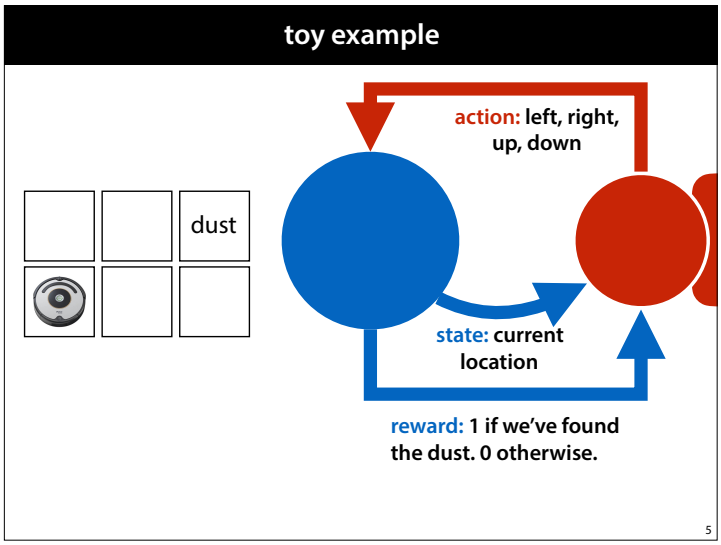


Reinforcement learning (RL) is an **abstract task**, and it is one of the most generic abstract tasks available.. Almost any learning problem you encounter can be modelled as a reinforcement learning problem (although better solutions will often exist).

The source of examples to learn from in RL is the environment. The agent finds itself in a **state**, and takes an **action**. In return, the environment tells the agent its new state, and provides a **reward** (a number, the higher the better). The agent chooses its action by a **policy**: a function from states to actions. As the agent interacts with the world the **learner** adapts the policy in order to maximise the expectation of future rewards.

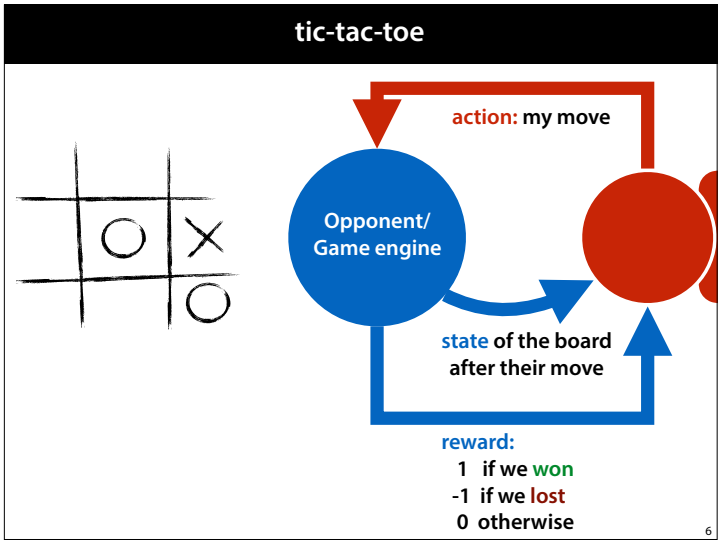
In order to translate your problem to the RL framework you must decide what your states and actions are, and how to learn the policy.

The only true constraint that RL places on your problem is that for a given state, the optimal policy may not depend on the states that came before. Only the information in the current state counts. This is known as a **Markov decision process**.



Here is a very simple example for the floor cleaning robot. The room has six positions for the robot to be in, and in one of these, there is a pile of dust. For now we, assume that the position of the dust is fixed, and the only job of the robot is to get to the dust as quickly as possible. Once the robot find the dust, the world is reset, and the robot is placed in a random position. This is not a very realistic example, but it pays to start very simple. (If you really wanted to solve this specific problem you would simply use A*)

The environment has six states (the six squares). The actions are: up, down, left and right. Moving to any state yields a reward of zero, except for the G state, in which case it gets a reward of 1.



Games can also be learned through RL. In the case of a perfect information, turn-based two player game like tic-tac-toe (or chess or Go). The states are simple board positions. The **available actions** are the moves the player is allowed to make. After an action is chosen, **the environment chooses the opponent's move**, and returns the resulting state to the agent. All states come with reward 0, except the states where the game is won by the agent (reward = 1) or the game is lost (reward -1). A draw also yields reward 0.

cart pole

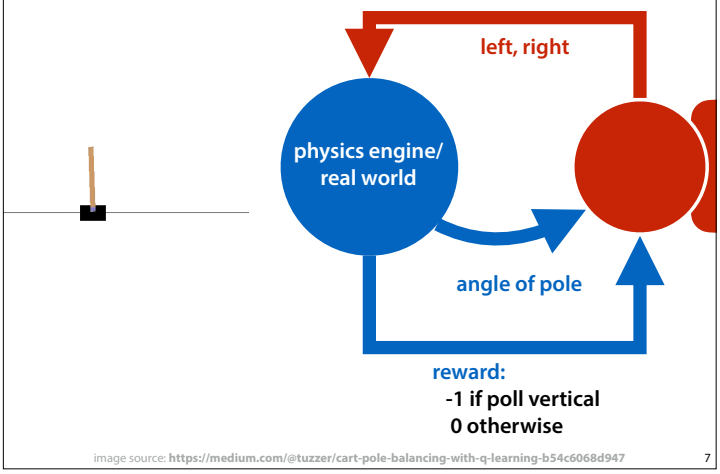


image source: https://medium.com/@tuzzer/cart-pole-balancing-with-q-learning-b54c6068d947

source: https://www.youtube.com/watch?v=VCdxqn0fcnE

Here is an example with a slightly faster rate of actions chosen: controlling a helicopter. The helicopter is fitted with a variety of sensors, telling it which way up it is, how high it is, it's speed and so on. The combined values for all these sensors at a given moment form the state. The actions following this state are the possible speeds of the main and tail rotor. The rewards, again, are zero unless the helicopter crashes, in which case it gets a negative reward. To train the helicopter to do specific tricks (like flying upside down), we can give certain states a positive reward depending on the trick

source: **https://www.youtube.com/watch?v=VCdxqn0fcnE**

sparse loss

Start with imitation learning:
Supervised learning, copying human action

Reward shaping:
Guessing the reward for intermediate states, or states near to good states.

Auxiliary goals:
Curiosity, maximum distance traveled

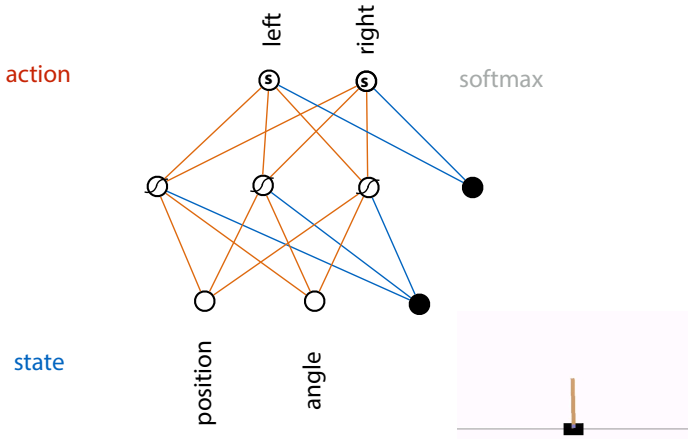
Good explanation of reward shaping: **https://www.youtube.com/watch?v=xManAGjbx2k**

One benefit of RL is that a single system can be developed for many different tasks, so long as the interface between the world and the learner stays the same. Here is a famous experiment by DeepMind, the company behind AlphaGo. The environment is an Atari simulator. The state is a single image, containing everything that can be seen on the screen. The actions are the four possible movements of the joystick and the pressing of the fire button. The reward is determined by the score shown on the screen.

The amazing thing here is that the system was not pre-programmed with any knowledge of any of the games. For several of the games the system learned play the game better than the top human performance.
source: ***https://www.youtube.com/watch?v=V1eYniJ0Rnk***

---

a policy network



11

Before we decide how to train our model, let's decide what it **is**, first. There are many ways to represent RL models, but most of the recent breakthroughs have come from using neural networks. Our job is to map states to actions, to states to a distribution over actions. We represent the state by two numbers (the position of the cart and the angle of the pole) and we use a softmax output layer to produce a probability distribution over the two possible actions.

If we somehow figure out the right weights, this is all we need to solve the problem: for every state, we simply feed it through the network and either choose the action with the highest probability, or sample from the outputs.

So now all we need is a way to figure out the weights.

---

the three problems of RL

Non differentiable loss

Balance exploration and exploitation

Delayed reward/sparse loss

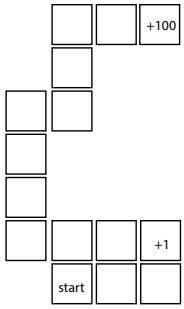
12

If your problem has any of these properties, it can pay to tackle in a reinforcement learning setting.

This can cause some confusion when the problem doesn't

exploration vs. exploitation

in state s, take action a



13

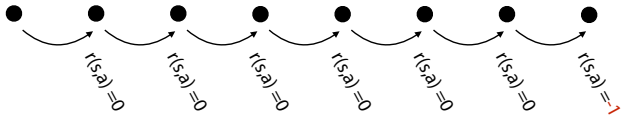
This is a classic trade-off in online learning: **exploration vs. exploitation**. Look at this scenario. Each time the agent finds a reward it is reset to the start state.

An agent stumbling around randomly will most likely find the reward top right first. After a few resets it has a good policy to reach that states. If it exploits only the things it has learned so far, it will keep coming back for the +1 reward, never-ending the +100 reward at the end of the long tunnel. An agent that follows a more random policy will explore more and eventually find the bigger treasure. At first, however, the exploring agent does markedly worse than the exploiting agent.

There is no definite answer to how to optimise this tradeoff, although a few best practices exist.

---

credit assignment problem



14

The main problem in reinforcement learning, is that we have to decide on our immediate *action*, but we don't get immediate *feedback*. If the pole falls over, it may be because we made a mistake 20 timesteps ago, but we only get the (negative) reward when the pole finally tips over. Once the pole started tipping over to the right, we may have move right twenty times: these were good actions, that should be rewarded, they were just too late to save the situation.

Another example is crashing a car. If we're learning to drive, this is a bad outcome that should carry a negative reward. However most people brake just before they crash. These are good actions that led to a bad outcome. We shouldn't learn not to brake before a crash, we should work backward to where we went wrong (lik taking a turn at too high a speed) and apply the negative feedback to those actions.

This is what's called the **credit assignment problem**, and it's what reinforcement learning is all about.

---

notation

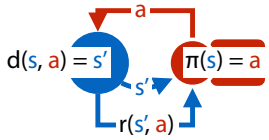
reward function:
$r(s, a) = 0.01$ (the higher the better)

state transitions
$d(s, a) = s'$

policy
$\pi(s) = a$ or $p(a|s) = 0.2$

the environment

our model

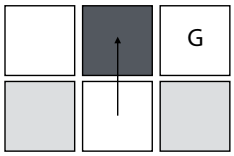


15

Here is some basic notation for the elements of Reinforcement learning. In most cases, the agent will not have access to the reward function, or the transition function and it will have to learn them. Sometimes the agent will learn a deterministic policy, where every state is always followed by the same action. In other cases it's better to learn a **probabilistic policy** where all actions are possible, but certain ones have a higher probability.

also possible

probabilistic state transitions



partially observable states

16

Here are some extensions to RL that we won't go into (too much) today.

Sometimes the state transitions are probabilistic. Consider the example of controlling a robot: the agent might tell its left wheel to spin 5 mm, but on a slippery floor the resulting movement may be anything from 0 to 5 mm.

Another thing you may want to model is partially observable states. For example, in a poker game, there may be five cards on the table, but three of them might be face down.

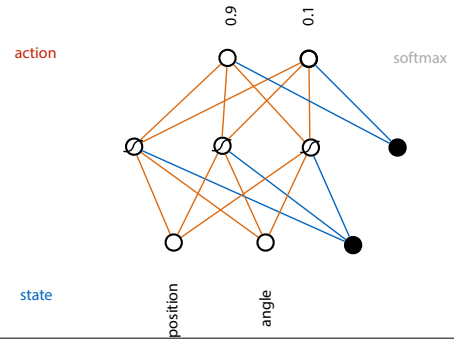
image source: ***http://www.dwaynebaraka.com/blog/2013/10/03/why-most-csr-budgets-are-wasted/***

---

choosing the weights (aka learning)

reward = 0

"move left"



17

Simple backpropagation doesn't work, because we don't have labeled examples that tell us which move to take for a given state. All we can do, is choose a move and observe the reward. And as mentioned, a big reward (positive or negative) is usually a response to the action chosen a while ago instead of the current action.

So how do we turn this into a way to update our weights?

---

random search

pick a random point **m** in the model space

loop:

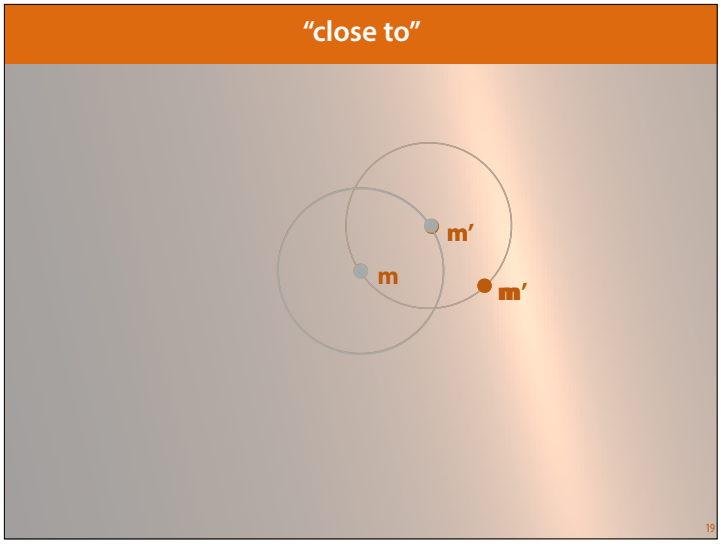
  pick a random point **m'** close to **m**

  if loss(**m**) < loss(**m'**):

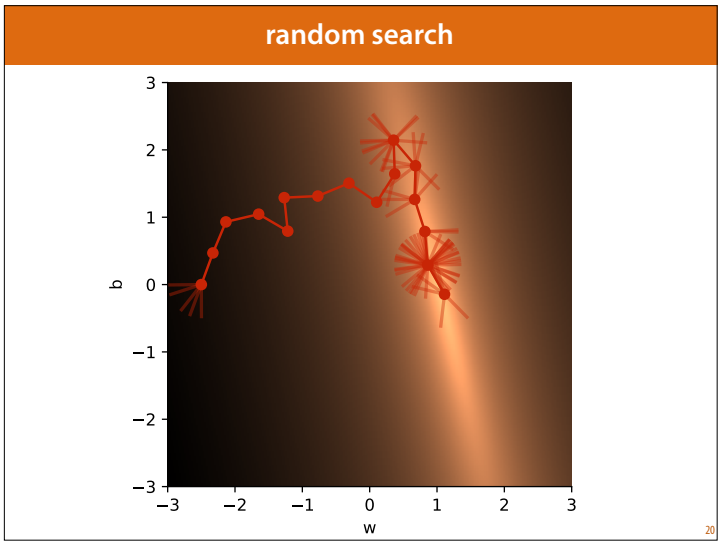
    m <- **m'**

18

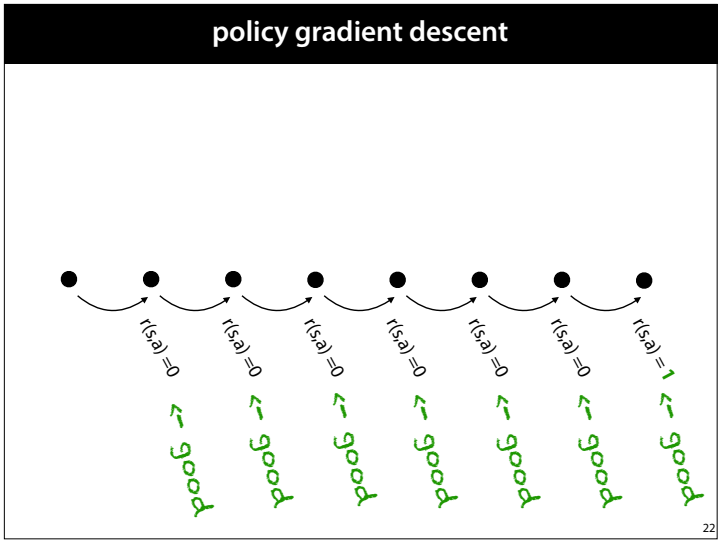
Let's start with a very simple example: random search.



The basic random search algorithm chooses the next point by sampling uniformly among all points with some pre-chosen distance r from w. Formally: it picks from the hypersphere (or circle, in 2D) with radius r, centered on w.

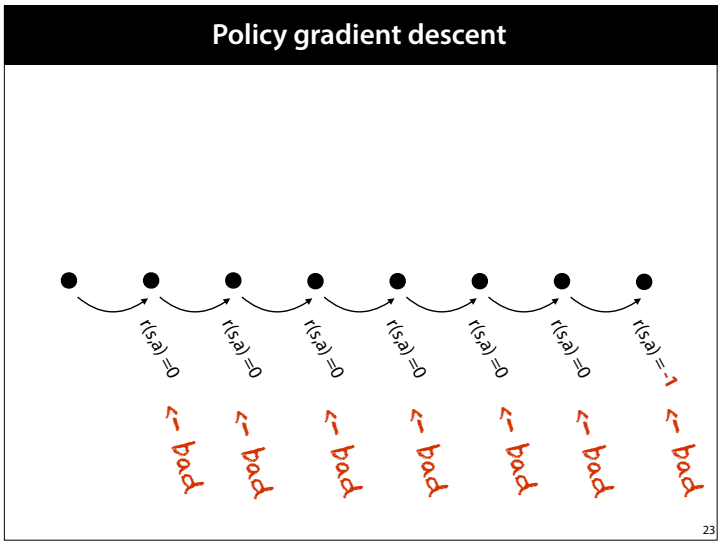


Here is random search in action. The transparent red offshoots are successors that turned out to be worse than the current point. The algorithm starts on the left, and slowly (with a bit of a detour) stumbles in the direction of the low loss region.

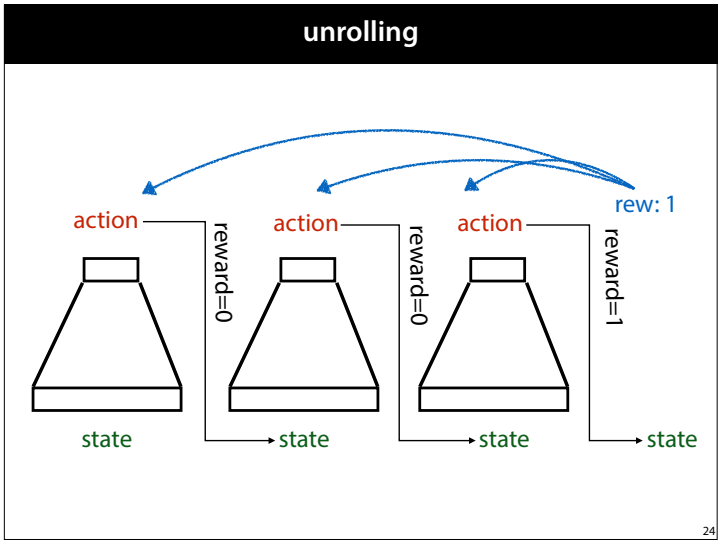


Here is a very simple solution that works well in situations where we have long sequences of actions providing no reward, followed by a single state providing a reward, after which the world is reset (for example playing tic-tac-toe). We simply follow some semi-random policy, wait until we reach a reward state, and then label all preceding state action pairs with the final outcome.

The idea is that if some of these actions were bad, on average they will occur more often in sequences ending with a negative reward, and on average they will be labeled bad more often than good.



In the case of the car crash, we should make sure the agent investigates the sequences where it doesn't brake before a crash as well (preferably in a simulated environment). Averaging over all sequences, braking before the crash results is less damage than not braking so the agent will eventually learn that braking is a good idea. Of course, we also have to make sure the reward is scaled according to the severity of the crash.



We can't back propagate over the unrolled network, because we are sampling the actions form the output.

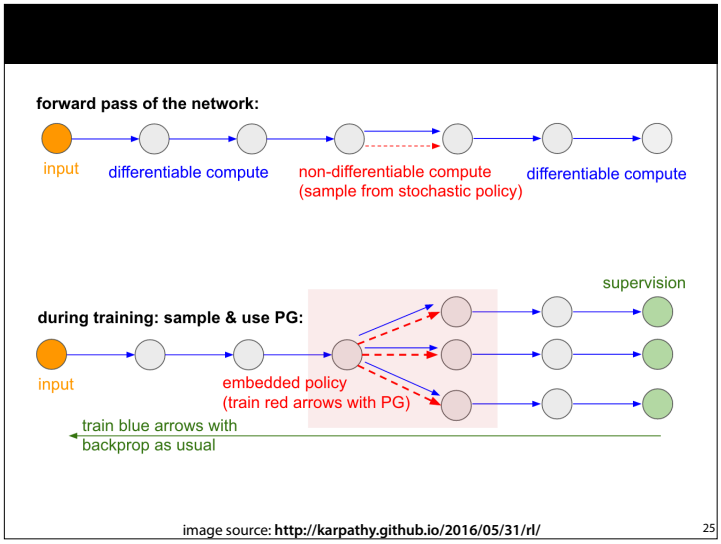


image source: http://karpathy.github.io/2016/05/31/rl/

policy gradients: the math

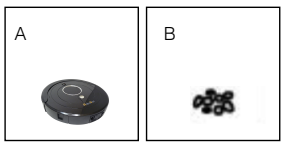
$$\begin{aligned}\nabla \mathbb{E}_a r(a) &= \nabla \sum_a p(a) r(a) \\ &= \sum_a \nabla p(a) r(a) \\ &= \sum_a p(a) \frac{\nabla p(a)}{p(a)} r(a) \\ &= \sum_a p(a) \nabla \ln p(a) r(a) \\ &= \mathbb{E}_a r(a) \nabla \ln p(a)\end{aligned}$$

$$\nabla \ln(z) = \frac{1}{z} \nabla z$$

26

Note r is not the immediate reward but the ultimate reward at the end of the trajectory.

Q-Learning



27

While policy gradient descent is a nice trick, it doesn't really get to the heart of reinforcement learning. To understand the problem better let's look at Q-learning, which is what was used in the Atari challenge.

The example we'll use is the robotic hoover, also used in the first lecture. We will make the problem so simple that we can write out the policy explicitly: The room will have two states, the hoover can move left or right, and one of the states has dust in it. Once the hoover finds the dust, we reset. (The robot is reset to state A, and the dust is replaced, but the robot keeps its learned experience).

what do we want to optimize?

discounted reward:

$r(s_0, a_0) + \gamma\, r(s_1, a_1) + \gamma^2\, r(s_2, a_2) + \gamma^3\, r(s_3, a_3) + \dots$

with $\gamma = 0.99$

or something similarly close to 1

28

If we fix our policy, then we know for a given policy what all the future states are going to be, and what rewards we are going to get. The discounted reward is the value we will try to optimise for: we want to find the policy that gives us the greatest discounted reward for all states. Note that this can be an infinite sum.

Note also that we are limiting ourselves here to deterministic policies: for a fixed policy we always do the same thing in the same state.

If our problem is finished after a certain state is reached (like a game of chess) the discounted reward has a finite number of terms. If the problem can (potentially) go on forever (like the cart pole) the sum has an infinite number of terms. In that case the discounting ensures that the sum still converges to a finite value.

---

definitions

policy: $\pi(s)$

value function:

$V^{\pi}(s_0) = r(s_0, a_0) + \gamma\, r(s_1, a_1) + \gamma^2\, r(s_2, a_2) + \dots$

$V^{\pi}(s_0) = r(s_0, a_0) + \gamma\, V^{\pi}(s_1)$

optimal policy:

π^* : the π such that for all states s, $\pi^* = \underset{\pi}{\text{argmax}}\, V^{\pi}(s)$

optimal value function:

$V^*(s) = V^{\pi^*}(s)$

29

The discounted reward we get from state s for a given policy π is called $V^{\pi}(s0)$, the value function. This represents the **value** of state s: how much we like to be in state s, given that we stick to policy π.

Using the value function, we can define our optimal policy, π*. This the policy that gives us the highest value function for all states. Note that this is always possible if policy A gives us the maximal value in state s but not in state q, and policy B gives us the maximal value in state q but not in state s, we can define a new policy that follows A in state s and B in state q.

We can then define V*(s), which is just the value function for the optimal policy.

---

definitions

$\pi^*(s) = \underset{a}{\text{argmax}}\, [\text{ "discounted reward of } V^*\text{" }]$

$\pi^*(s) = \underset{a}{\text{argmax}}\, [\, r(s, a) + \gamma\, V^*(d(s, a))]$

$Q^*(s, a) = r(s, a) + \gamma\, V^*(d(s, a))$

$\pi^*(s) = \underset{a}{\text{argmax}}\, Q^*(s, a)$

$V^*(s) = \underset{a}{\max}\, Q^*(s, a)$

30

Using V* we can rewrite π* as a *recursive* definition. The optimal policy is the one that chooses the action which maximises the future, assuming that we follow the optimal policy. We fill in the optimal value function to get rid of the infinite sum. We've now defined the optimal policy in a way that depends on what the optimal policy is. While this doesn't allow us to compute π*, it does *define* it. If someone gives us a policy, we can recognise it by checking if this equality holds.

To make this easier, we take the part inside the argmax and call it Q(s, a). We then rewrite the definitions of the optimal policy and the optimal value function in terms of Q(s,a).

How has this helped us? Q(s,a) is a function from state-action pairs to a number expressing how good that particular pair is. If we were given Q, we could automatically compute the optimal policy, and the optimal value function. And it turns out, that in many problems it's much easier to learn the Q-function, than it is to learn the policy directly.

making the definition of Q* recursive

$$Q^*(s, a) = r(s, a) + \gamma V^*(d(s, a))$$

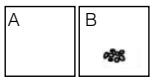
$$Q^*(s, a) = r(s, a) + \gamma \max_{a'} Q^*(d(s, a), a')$$

31

In order to see how the Q function can be learned we rewrite it. Earlier, we rewrote the V functions in terms of the Q function, now we plug that definition back into the Q function. We now get a recursive definition of Q.

Again, this may be a little difficult to wrap your head around. If so think of it this way: If we were given a random Q-function, how could we tell whether it was optimal? We don't know π* or V* so we can't use the original definitions. But this equality must hold true! If we loop over all possible states and actions, and plug them into this equality, we must get the same number on both sides. Let's try it for a simple example.

Is my Q-function optimal?



r(A, R) = 1, all others 0

s	a	r(s,a)	Q(s,a)
A	L	0	1
A	R	1	2
B	L	0	1
B	R	0	-1

$$Q(s, a) \stackrel{?}{=} r(s, a) + \gamma \max_{a'} Q(d(s, a), a')$$

32

This is the two-state hoover problem again. We have states A and B, and actions left and right. The agent only gets a reward when moving from A to B. On the bottom left we see some random policy, generated by assigning random numbers to each state action pair. Did we get lucky and stumble on the optimal policy? Try it for yourself and see. (take γ = 0.9)

solving recurrent equations by *iteration*

$$x = x^2 - 2$$

$$x = 0:$$

$$0 \rightarrow 0^2 - 2 = -2$$

$$x = -2:$$

$$-2 \rightarrow -2^2 - 2 = 2$$

$$x = 2:$$

$$2 \rightarrow 2^2 - 2 = 2$$

33

Of course, random sampling of policy functions is not an efficient search method. How do we get from a recursive definition to the value that satisfies that definition? Here is a simple example from a single number: define x as the value for which $x = x^2 - 2$ holds. This is analogous to the definition above: we have one x on the left and a function of x on the right.

Of course, we all learned in high school how to solve this by rewriting, but we can also solve it by *iteration*. We replace the equals sign by an arrow and write: $x \leftarrow x^2 - 2$. We start with some randomly chosen value of x, compute $x^2 - 2$, and replace x by the new value. We iterate this and we end up with a functions for which the definition holds. Try it for yourself (start with x = 0)

Note that in this example infinity also counts as a solution, so if you pick the wrong starting state you may end up with larger and larger numbers. For other functions, there may be stable states that jump back from one point to another, or even chaotic states (see **https://en.wikipedia.org/wiki/Logistic_map** for more information if you're interested, but this is not exam material).

Q-Learning

init Q(s, a) = 0 for all s and a

loop:

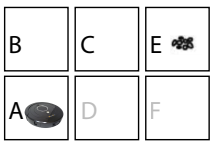
- in state s, take action a
  - arrive in s'
  - receive reward r
- update $Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a')$

34

This gives us the Q-learning algorithm shown here.

Note that the algorithm does not tell you how to choose the action. It may be tempting to use your current policy to choose the action, but that may lead you repeat early successes with out learning much about the world.

NB:While we are learning a deterministic policy here (the Q function), the function that decides which actions to take can be anything, and should contain some randomness.



$Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a')$

s	a	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
			A U B R C R E			A U B R C R E		
A	U	0	0	0	0	0	0	0
A	R	0	0	0	0	0	0	0
B	U	0	0	0	0	0	0	0
B	R	0	0	0	0	0	0	0
C	U	0	0	0	0	0	0.9	0.9
C	R	0	0	0	1	1	1	1
D	U	0	0	0	0	0	0	0
D	R	0	0	0	0	0	0	0
E	U	0	0	0	0	0	0	0

reset

35

To see how Q learning operates, imagine setting a robot in the bottom-left square (A) in the figure shown and letting it explore. The robot chooses the actions up, right, right and when it reaches the goal state (E) it gets reset to the start state. It gets +1 immediate reward for entering the goal state and 0 reward for any other action.

What we see is that the Q function stays 0 for all values until the robot enters the goal state. At that point Q(C, R) west updated to value one. In the next run, Q(B, R) gets updated to 0.9. In the next run after, Q(A, U) is updated to 0.9 * 0.9. This is how Q-learning updates. In every run of the algorithm the immediate rewards from the previous runs are propagated to neighbouring states.

Q-Learning

Which actions should we take?

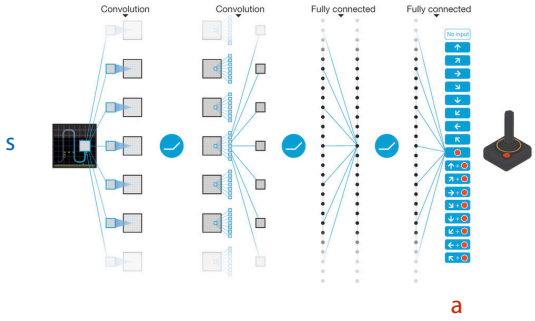
epsilon-greedy: follow current pollicy, expect with probability epsilon, take a random action

Decay epsilon as learning progresses

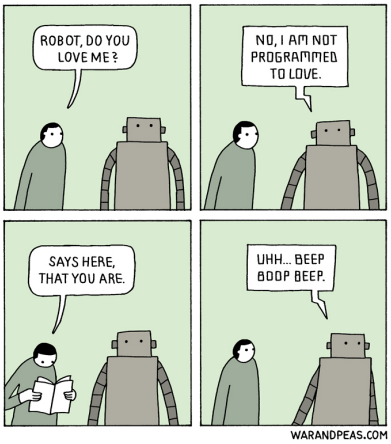
36

Deep Q-learning

update Q(s, a) <- r + $\gamma \max_{a'} Q(s', a')$



break



source: **https://warandpeas.com/2016/10/09/robot/**

AlphaGO

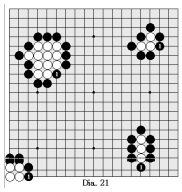
source: **https://www.youtube.com/watch?v=8tq1C8spV_g**

Image: Lee Jin-man/ AP

In 2016 AlphaGo, a Go playing computer developed by the company DeepMind beat the world champion Lee Sedol. Many AI researchers were convinced that this AI breakthrough was at least decades away.

image source: **http://gadgets.ndtv.com/science/news/lee-sedol-scores-surprise-victory-over-googles-alphago-in-game-4-813248**

The game of Go


Dia. 21

41

First, some intuition about how Go works. The rules are very simple: players (black and white) move, one after the other, placing stones on a 19 by 19 grid. The aim of the game is to have as many stones on the board, when no more stones can be placed. The only way to remove stones is to encircle your opponent. Why is Go so difficult and what has AlphaGo done to finally solve these issues?

claims from the media

AlphaGo is an important move towards general purpose AI.

AlphaGo thinks and learns in a human way.

AlphaGo mimics the human brain.

Go has more possible positions than there are atoms in the universe. That's why it's difficult.

What makes Go difficult is the high branching factor.

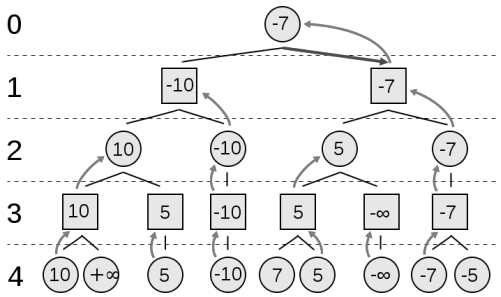
42

When the win against Lee Sedol was publicised, many claims were made in the media, some by DeepMind themselves. Here are some of them. All of these are dubious for various reasons

From top to bottom:

- AlphaGO is very much purpose-built for Go. It's not an architecture that can be translated 1-on-1 to any other games, and Go has some features that are exploited in a very specific way. However, DeepMind hasher projects that *are* impressive milestones toward general purpose AI. It's also true that projects like Deep Blue (the chess computer that beat Kasparov) were filled with hand-coded chess knowledge, written with the help of experts. This is not true for AlphaGo: the rules of Go were hardcoded into it, and it learned everything else by simply observing existing matches, and playing against itself
- AlphaGo learns. It's thinking is probably more human than Deep Blue's, but we don't understand human thinking well enough to make this claim.
- AlphaGo uses convolutional neural networks, which very loosely inspired by brain architecture.

minimax



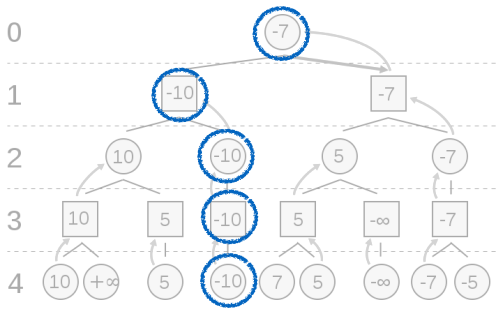
43

The minimax algorithm is mostly useless when it comes to Go. For each node in the tree there are up to 361 children, compared to about 30 for chess. This means almost 17 billion terminal nodes if we just search two turns deep. And as we discussed, you need to search very deep to find the nodes that show clear rewards.

image source: By Nuno Nogueira (Nmnogueira) - http://en.wikipedia.org/wiki/Image:Minimax.svg, created in Inkscape by author, CC BY-SA 2.5, ***https://commons.wikimedia.org/w/index.php?curid=2276653***

---

rollouts

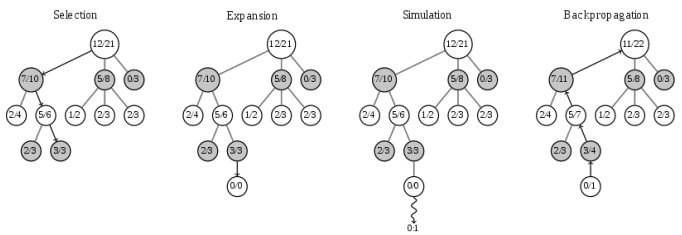


44

This simple principle was an early success in playing Go: we simply choose random moves from some fast policy, and play a few full games for each immediate successor. We then average the rewards we got over these as a value for the successor states, and choose the action that lead to the highest values. The **rollout policy** should ideally give good moves high probability, but also be very fast to compute.

---

monte-carlo tree search (MCTS)



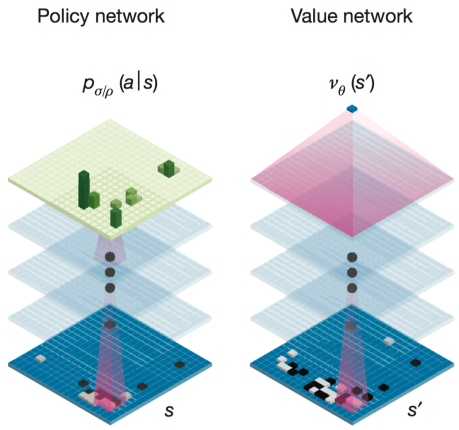
45

Monte Carlo Tree Search (MCTS) is a simple, but effective algorithm, combining rollouts with an incomplete tree search. We start the search with an unexpanded root node labeled 0/0 (this value represents the probability of winning from the given state). We then iterate the following algorithm

- Selection: select unexpanded node. At first, this will be the root node. But once the tree is further expanded we perform a random walk from the root down to one of the leaves.
- Expansion: Once we hit a leaf (an unexpanded node), we expand it and give its children the value 0/0.
- Simulation: From each expanded child we do a rollout.
- Back propagation (nothing to do with the backpropagation we know from NNs): If we win the rollout let v = 1 otherwise v = 0. For the new child and everyone of its parents update the value. If the old value was a/b, the new value is a+v / b+1. The value is the proportion of simulated games crossing that state that we've won.

The random walk performed in the selection phase should favour nodes with a high value, but also explore the nodes with

AlphaGo (2016)



46

the functions that AlphaGo learns are convolutional networks. One type is a policy network (from states to moves) and one is a value network (from states to a numeric value).

Here are the networks it learns

SL: policy network: from a database of games (like ALVINN, watching human drivers)

RL: policy network, start with SL, but refine with reinforcement learning (policy gradient descent)

slow policy network (with softmax layer on top)

fast policy network with (with linear activations)

V: value network learned from observing older versions of itself playing games. Once the game is finished and the outcome of the game (eg. "black wins") is used as the label for all the states observed in the game.

The value network predicts the winner form the current board state.

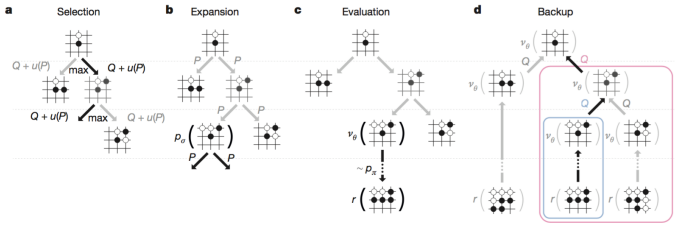
---

putting it all together

start with imitation learning.

improve by playing against previous iterations and self
update weights by policy gradients

Boost network performance by MCTS



The networks are trained by reinforcement learning using policy gradient descent.

During actual play, AlphaGO uses an MCTS algorithm. The value on each node (as in basic MCTS) represents the probability that black will win from that state.

- When it comes to rollouts, use the slow policy network to do a rollout for T steps, then finish with the fast policy
- The value v of the newly opened state is the average of the value (computed with the the value network) after T steps, and the win/lose value at the end of the rollout. (image c)
- Backup as with standard MCTS: each node's value becomes the probability of a win from that state. Precisely the value of node n becomes the sum of the values of all simulations crossing node n, divided by the total number of simulations crossing node n. A simulation is counted as a full game simulated from the root node down to a terminal (win/loss) node. (image(d))

AlphaGo Zero (2017)

Learns from scratch, no imitation learning, reward shaping etc.
Also applicable to Chess, Shogi…

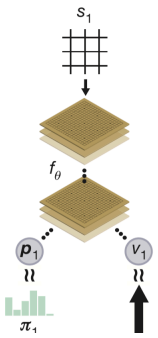
Uses three tricks to simplify/improve AlphaGo

1. Combine policy and value nets
2. View MCTS as a *policy improvement operator*
3. Add residual connections, batch normalization

48

AlphaGo Zero (2017)

trick 1: combine policy and value nets.



AlphaGo Zero (2017)

trick 2: view MCTS as a *policy improvement operator*.

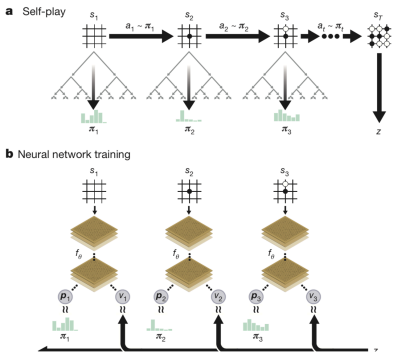


image source: Mastering the game of Go without human knowledge, David Silver, Julian Schrittwieser, Karen Simonyan et al.

AlphaGo Zero (2017)

trick 2: view MCTS as a *policy improvement operator*.

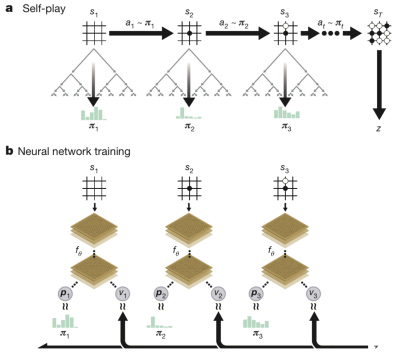
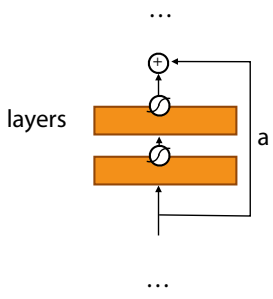


image source: Mastering the game of Go without human knowledge, David Silver, Julian Schrittwieser, Karen Simonyan et al.

AlphaGo Zero (2017)

trick 3: residual connections and batch normalisation



52

initialization (deep learning 1)

Make sure your data is standardized/normalized
i.e. the mean is close to **0**, and the variance is close to 1 in every direction.

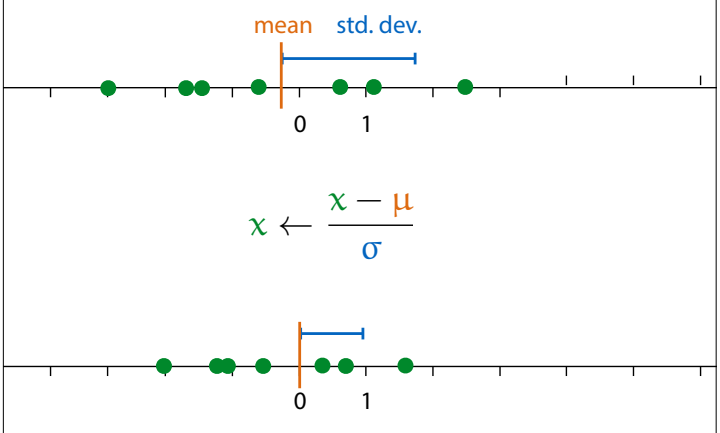
initialise weights W:

- Make W a random orthogonal matrix (eigenvalues all 1).
- Glorot uniform:

$$w_{ij} \sim U\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$$

53

standardisation (Methodology 2)

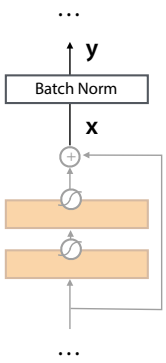


54

Another option is standardization. We rescale the data so that the mean becomes zero, and the standard deviation becomes 1. In essence, we are transforming our data so that it looks like it was sampled from a standard normal distribution (as much as we can with a one dimensional linear transformation).

We can think of the data as being generated from a standard normal distribution, followed by multiplication by sigma, and and adding mu. The result is the distribution of the data. If we then compute the mean and the standard deviation of the data, the formula in the slide is essentially inverting the transformation, recovering the "original" data as sampled from the normal distribution.We will build on this perspective to explain whitening.

batch normalisation



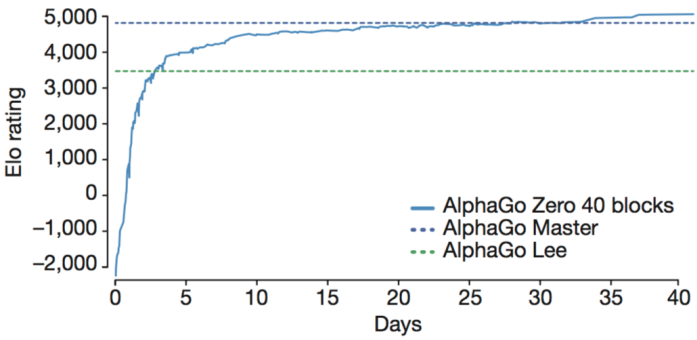
$x_1, \ldots, x_m$: output of previous layer
$y_1, \ldots, y_m$: result
γ, β : learnable parameters

$$\mu = \frac{1}{m} \sum x_i \qquad \text{mean over batch}$$

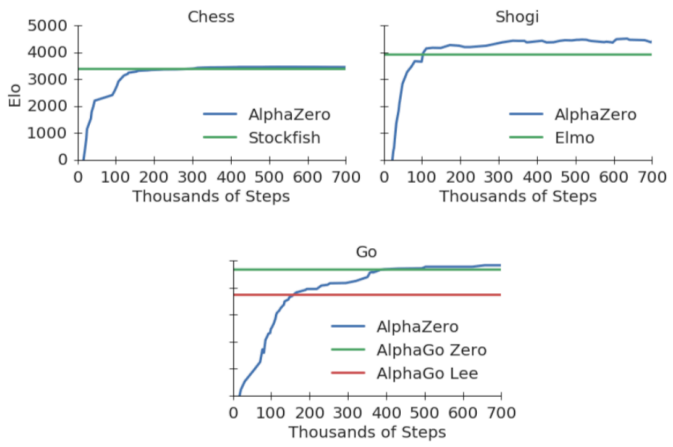
$$\sigma = \sqrt{\frac{1}{m} \sum (x_i - \mu)^2} \qquad \text{std. dev. over batch}$$

$$\hat{x}_i = \frac{x_i - \mu}{\sigma + \epsilon} \qquad \text{standardize}$$

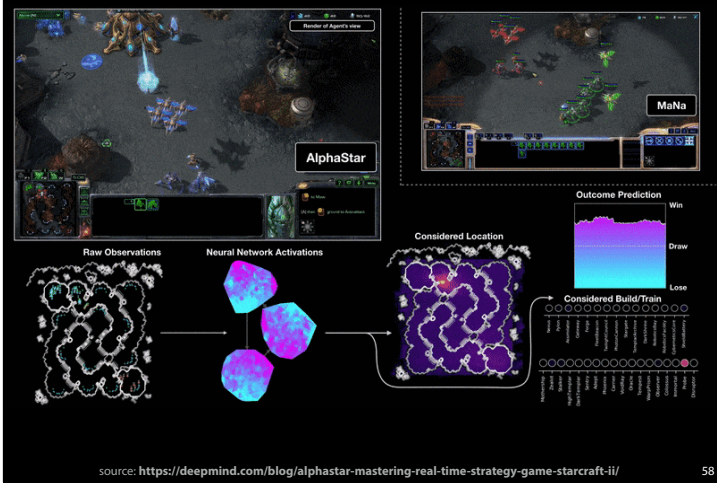
$$y^i = \gamma \hat{x}_i + \beta$$



Alpha Zero (2017)



AlphaStar



source: **https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/**

Starcraft

Real time
No "searching the game tree"

Imperfect information
Use scouting to trade units against information

Large, diverse action space
Hundred of units and building with many possible actions

No single best strategy

How AlphaStar is trained

AlphaStar's behaviour is generated by a deep **neural network** that receives input data from the raw game interface (a list of units and their properties), and outputs a sequence of instructions that constitute an action within the game. More specifically, the neural network architecture applies a **transformer** torso to the units (similar to **relational deep reinforcement learning**), combined with a **deep LSTM core**, an **auto-regressive policy head** with a **pointer network**, and a **centralised value baseline**. We believe that this advanced model will help with many other challenges in machine learning research that involve long-term sequence modelling and large output spaces such as translation, language modelling and visual representations.

transformer torso for the units

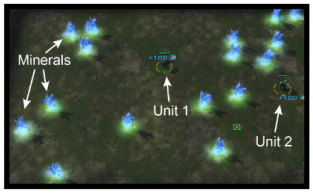
deep LSTM core with

- autoregressive policy head
- pointer network

multi-agent learning

transformer: relational inductive bias

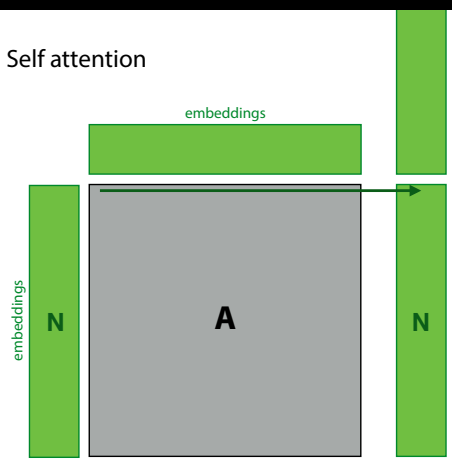
StarCraft II mini-game
Collect Mineral Shards



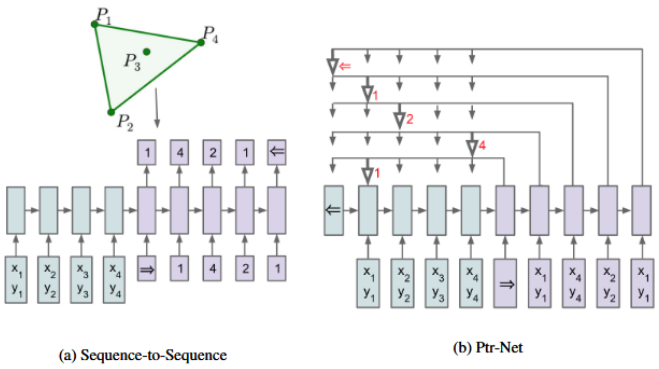
StarCraft II mini-game
Build Marines

transformer

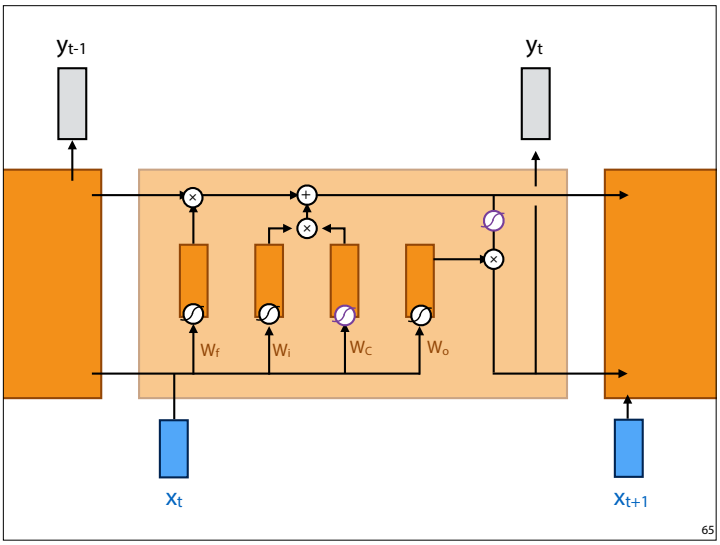
Self attention



pointer networks



64



65

Here is what happens inside the cell. It looks complicated, but we'll go through all the elements step by step.

sequential sampling from a language model

start with a small *seed* sequence s = [c_1, c_2, c_3] of tokens.

loop:

Sample next char c according to p(C = c | c_1, c_2, …)
feed the *whole* seed to the network

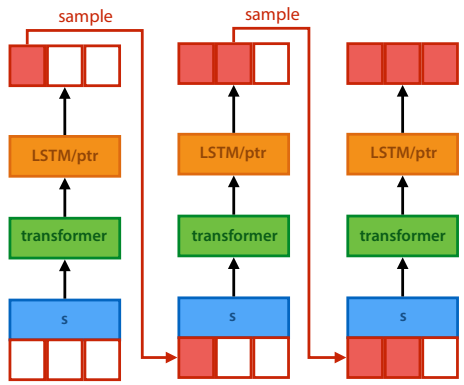
append c to s

also known as a *autoregressive RNN*

66

Note that this time, there is no Markov assumption. The network has to see the whole sequence so far to predict the next character.

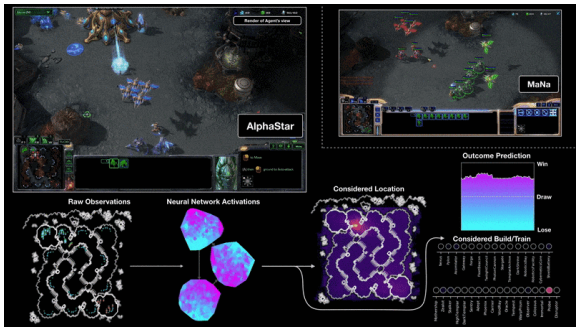
autoregres over actions



multi-agent learning

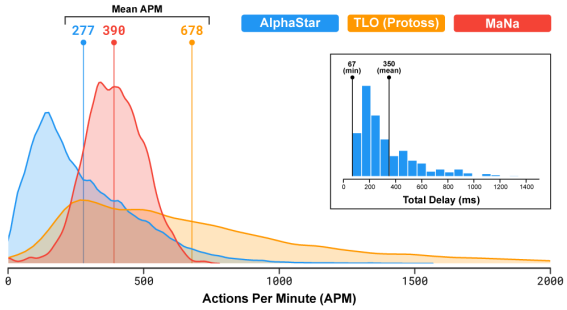
does AlphaStar have a physical advantage?

issue 1: the global camera



does AlphaStar have a physical advantage?

issue 2: actions-per-minute



mlcourse@peterbloem.nl